The implementation of AKL(FD)

Björn Carlson[1]
Mats Carlsson
Sverker Janson

Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden
{bjornc, matsc, sverker}@sics.se

Abstract

AKL(FD) is an integration of (an extension of) the finite domain constraint system FD in AKL, a deep concurrent constraint programming language [CJH94]. In this paper we describe the implementation of the FD solver and its adaptation to the generic constraint interface of the AGENTS implementation of AKL. We also describe compilation techniques used by the AKL(FD) compiler.

1 Introduction

Conceptually, a concurrent constraint programming (CCP [SRP91]) language is an ideal vehicle for constraint programming. The expressiveness of the constraint solver can be extended by user-defined entailment-driven propagation rules that execute concurrently and cooperate with the constraint solver. However, to offer an orthogonal combination of constraint programming with the other paradigms offered by CCP—concurrent, relational, functional, object-oriented, ...—the language must be *deep*. Being deep means having a *hierarchy* of constraint stores, where a computation need not be affected by the failure of a subordinate store. This makes it possible to have a reactive process/object-oriented top-level in a program, with other *encapsulated* components performing constraint solving.

AKL is a deep CCP language [JH91, Jan94] and AKL(FD) is an integration of (an extension of) the indexical-based finite domain constraint system FD [VHSD91] in AKL [CJH94, Car95]. That AKL(FD) is very expressive and that good performance can be achieved has been argued elsewhere [CJH94]. The purpose of the present paper is to give the details of the implementation of the FD solver, which operates in a hierarchy of constraint stores, and its adaptation to the generic constraint interface of the AGENTS implementation of AKL.

Our work has four main aspects: (1) Execution of indexicals is optimized. (2) High-level finite domain constraints, such as symbolic and propositional

[1] Computing Science Department, Uppsala University, Box 311, 751 05 Uppsala

constraints, are compiled. (3) The FD solver is integrated into AGENTS using the generic constraint interface, making it a plug-in module. (4) The copying scheme for nondeterminism in AGENTS is complemented with a simple but effective sharing scheme.

Optimizations that we exploit are minimizing reexecutions of indexicals, propagating in constant space, locating indexicals in the hierarchy of stores, and checking entailment efficiently. These are described in the paper. In particular, the FD solver uses an algorithm which, by regarding the monotonicity of indexicals, can check the entailment of indexicals as efficiently as the consistency. Hence, we are able to implement a simple scheme for entailment checking of finite domain constraints.

We compile arithmetic, symbolic and propositional constraints, such as disjunctions of finite domain constraints, either to indexicals or to (guarded) clauses, preserving the operational and denotational semantics of the constraints. In this paper we outline the compilation scheme for arithmetic constraints.

The AGENTS implementation of AKL inlines handling of tree constraints, as in the WAM, and supports other constraint solvers (and their pertaining data types) through a generic constraint interface. New constraint solvers (and data handlers) written in C can be added as plug-in modules to the system, without affecting existing code. This solution is more efficient than, e.g., atttributed variables, allowing more low-level representations and manipulations of constraints, but somewhat less efficient than an inlined constraint solver.

The major obstacle to obtain efficiency in AKL(FD) is that don't know nondeterminism in AGENTS is implemented by copying parts of the state [Jan94]. We have experimented with extending AGENTS with a trailing mechanism used for solving finite domain constraints, such that AKL(FD) keeps up with CHIP in performance for a set of standard benchmarks [CJH94].

Previously, FD has been implemented in the CLP framework, e.g., the clp(FD) system by Diaz and Codognet [DC93]. This system is orthogonal to ours in that it extends the execution engine for Prolog with support for FD, and compiles all source code to C, thus demonstrating that indexical-based constraint systems can be very efficient.

The language cc(FD) is derived from FD [VHSD92], and exploits high-level combinators for efficient constraint programming. In contrast, AKL(FD) uses the concurrent constraint paradigm to show that by efficient implementation of the indexicals, the high-level combinators of cc(FD) can be programmed in the language itself with reasonable efficiency [CJH94].

The rest of the paper is structured as follows. In Section 2 we explain the constraint store model of AKL and the generic constraint interface of

AGENTS. In Section 3 we give the instantiation of the constraint interface with FD. Section 4 outlines the FD solver, the FD emulator for evaluating ranges, and the trailing mechanism for FD added to AGENTS. Finally, Section 5 presents the FD compiler.

2 Operations on Constraints and Stores

We now describe the constraint store model of AKL and the generic constraint protocol of AGENTS.

2.1 Constraint Stores

Seen from a logical perspective, stores in an AKL computation state are of the following general form.

$$P ::= \exists V(\theta \wedge Q \wedge \cdots \wedge Q)$$

where each Q is of the form

$$Q ::= (P \vee \cdots \vee P)$$

For simplicity, let us call components of the P form *conjunctions* and expressions of the Q form *disjunctions*. Conjunctions contain *constraint stores* θ. Variables in V are said to be *local* in a conjunction.

Computations are performed by a worker moving about in the computation state, performing rewrites corresponding to the different possible transitions between states. The *environment* of a rewritten component is the set of stores θ in the conjunctions "surrounding" it. When moving in the computation state, the worker maintains an efficient representation of the current environment.

Non-determinism in AKL is obtained by splitting part of a computation state in two branches. This is done simply by replacing

$$\exists V(\theta \wedge (A \vee B) \wedge Q)$$

with

$$\exists V(\theta \wedge A \wedge Q) \vee \exists V(\theta \wedge B \wedge Q)$$

and applying suitable simplifications. The duplicated parts may be rewritten quite differently. Such differences can be maintained by trailing, but, for simplicity, the AGENTS implementation performs copying corresponding to the duplication of components in the above schematic rule.

In AKL, disjunctions are created by the execution of the various choice statements, and the conjunctions they contain correspond to the execution of guards. Disjunctions and conjunctions also occur at the top-level and in aggregates.

A guard computation in AKL is hence embedded within a private local store, and no distinction is made between execution inside or outside guards.

This fundamental design choice enables encapsulated (non-determinate) computations as well as logically more complete implementations. The latter follows simply because guards involving disentailed and mutually incompatible constraints can be discarded.

In the design of AGENTS we have adopted a view of constraints mimicking the constraint view of Prolog, i.e., we represent the constraints through the variables they constrain. Thus, from the implementation point of view, we consider the store not to be a set of constraints [SRP91], but a set of variables. Any constraint $c(X)$ adds its information on X, ensuring that X contains enough information to recover the meaning of c.

In the following we use C syntax in describing the outline of the structures involved. A constraint variable is thus layed out as:

```
struct {
  varMethod  *method;
  id         *env;
  suspension *susp;
}
```

where `method` is a vector of functions that may be applied to a variable, `env` is a tag identifying the locality of the variable, and `susp` is a list of predicate calls that are waiting for the variable to receive a value.

The function vector contains the following functions:

```
varMethod method =
{
  new,
  unify,
  print,
  copy,
  gc,
  copy_external,
  gc_external
}
```

Except for `copy_external` and `gc_external`, the purpose of the functions should be obvious. `copy_external` (`gc_external`) is called by the copying (garbage collecting) procedure to allow a constraint variable to duplicate any suspensions of copied goals.

2.2 Constraint operations

The (sequential) AGENTS worker is basically a transformer of configurations, where a *configuration* consists of a constraint store hierarchy, as above, and the occurrence of the interpreter. Each conjunction P refers to a list of

suspended constraints P_C. The environment of P is referred to as P_θ. The worker rewrites the configurations according to the operational semantics of AKL [JH91, Jan94].

Suppose a constraint c is executed in P. If c and P_θ are inconsistent, the execution fails and P is marked as dead. If c is entailed, the computation succeeds. Otherwise, c is *simplified* with respect to P_θ, and added to P_C. Thus, the execution of c terminates.

The worker exits a conjunction P when P_C is not entailed by P_θ, and the associated guard operator, such as $|$, of P expects entailment. Hence, P is *deinstalled*, i.e. P_C is retracted from P_θ. This requires that the representation of P_C is kept explicit.

The worker *installs* a conjunction P when some update to P_θ have affected the value of variables constrained in P_C. Hence, if P_C is inconsistent with P_θ, the installation fails, if P_C is entailed by P_θ the installation succeeds, and otherwise the installation suspends and the conjunction is deinstalled.

Suppose P occurs in

$$Q = (P_1 \vee \cdots \vee P \vee \cdots \vee P_k),$$

where Q occurs in a conjunction

$$P' = \exists W(\gamma \wedge Q_1 \wedge \cdots \wedge Q \wedge \cdots \wedge Q_l).$$

If Q is replaced with P in P', e.g. by committing to a certain guard, P is *promoted*, i.e. P' becomes

$$\exists W \cup V(\gamma \cup \theta \wedge Q_1 \wedge \cdots \wedge P \wedge \cdots \wedge Q_l)$$

(where we assume $W \cap V = \emptyset$) and P_C is added to P'_C. Variables previously external in P_C may now be local, since W and V have been merged, and thus constraints in P_C may become entailed by the promotion. The internal representation of the constraints can be optimized by the promotion, since information that was needed for deinstallation is now redundant and can thus be removed. Furthermore, P is marked as dead and linked to P' such that any object local in P becomes local in P'.

On the low level a constraint is represented as:

```
struct constraint {
  constraintMethod  *method;
  struct constraint *next;
}
```

The `next` field is used to link constraints into the list of locally declared constraints. The `method` field is a function vector which contains the following functions:

$$
\begin{array}{rcl}
N & ::= & x \mid i, \text{ where } i \in \mathcal{N} \\
T & ::= & N \mid T + T \mid T - T \mid T * T \mid T/T \mid T \textbf{ mod } N \\
 & \mid & \textbf{min}(N) \mid \textbf{max}(N) \\
R & ::= & R \wedge R \mid R \vee R \mid R \Rightarrow R \mid R \setminus R \\
 & \mid & R + R \mid R - R \mid R * N \mid R/N \mid R \textbf{ mod } N \\
 & \mid & T..T \mid \textbf{dom}(N)
\end{array}
$$

Figure 1: Syntax of range expressions

```
constraintMethod method = {
  install,
  reinstall,
  deinstall,
  promote,
  print,
  copy,
  gc
}
```

where the meaning of the functions should be obvious, except for the `reinstall` function which is called after copying or garbage collecting parts of the memory. *Reinstallation* is a special case of installation where the installed constraints cannot fail nor propagate.

3 FD integration

We now explain the details of the internal structures used for representing FD indexicals and variables. First, we give a brief outline of FD. For more information see [Car95].

3.1 FD

The constraint system FD is based on *domain constraints* and functional rules called *indexicals* [VHSD91]. A domain constraint is an expression $x \in I$, where I is a finite set of natural numbers. A set S of domain constraints is called a *store*. x_S is defined as the intersection of all I such that $x \in I$ belongs to S (if no such $x \in I$ belongs to S, $x_S = Z$). An indexical has the form x **in** r, where r is a *range* (generated by R in Figure 1). When applied to a store S, x **in** r evaluates to a domain constraint $x \in r_S \cap \mathcal{N}$, where r_S is the value of r in S (see below).

The value of a range r in S, r_S, is a set of integers computed by the set functions defined by R in Figure 1. For example, the expression $\textbf{dom}(y)$ evaluates to the set to which y is constrained in S. The expression $t_1..t_2$ computes the interval between t_1 and t_2, and the operators $\vee$, $\wedge$ and $\setminus$

x_S related to r_S	r monotone	r antimonotone
$x_S \cap r_S = \emptyset$	inconsistent	may become entailed
$x_S \subseteq r_S$	may become inconsistent	entailed
$x_S \neq (x_S \cap r_S) \neq \emptyset$	may become inconsistent	may become entailed

Table 1: Entailment/Inconsistency of x **in** r in a store S

denote union, intersection and set difference respectively (thus overloading the meaning of the symbols). The conditional range $r \Rightarrow r'$ equals r'_S if $r_S \neq \emptyset$ and $\emptyset$ otherwise. The expressions $r + t$, $r - t$, and r **mod** t denote the integer operators applied pointwise. The expression $-r$ is used instead of $0..k \setminus r$ for some large constant k.

The value of a term t in S is defined in the obvious way, where the expressions $\mathbf{min}(r)$ and $\mathbf{max}(r)$ evaluate to the infimum and supremum values of r_S.

A range r is *monotone* if for every pair of stores S_1 and S_2 such that S_2 implies S_1, $r_{S_2} \subseteq r_{S_1}$, and r is *antimonotone* if for every pair of stores S_1 and S_2 such that S_2 implies S_1, $r_{S_1} \subseteq r_{S_2}$.

The consistency and entailment of x **in** r in a store S is checked by considering the relationship between x_S and r_S, together with the monotonicity of r (see Table 1). In particular, suppose an indexical x **in** r is executed in store S, and suppose further that r is monotone. Hence, if $x_S \cap r_S \neq \emptyset$, $x \in r_S$ is added to S since any assignment n, consistent with x **in** r, of x must be such that $n \in x_S \cap r_S$. If $x_S \not\subseteq r_S$, we say that x is *pruned*.

3.2 FD structures

There are three main FD structures, i.e. indexicals, variables, and constraints.

3.2.1 Indexicals An indexical X **in** r is represented as:

```
struct fd_indexical {
  Term x;
  long c;
  unsigned long info;
  id   *env;
  Term ent;
  argvec args;
  monstruct *moninfo;
}
```

where `x` is a reference to X, `c` is an index to the byte code representing r, `info` contains information about whether the indexical is used for entailment checking, whether the indexical is part of an inconsistent local

store, and when the indexical was last executed. `env` is the identity of the store in which the indexical was added, `ent` is a flag set to true when the indexical is entailed, `args` is a vector of the arguments to the indexicals, and `moninfo` contains decision information used for deciding whether the indexical is monotone or antimonotone [CCD94].

A range is compiled into a postfix notation, which is then translated straightforwardly into byte code by the loader. The idea is simple: a range expression R_1 Op R_2 is translated into "code(R_2) code(R_1) instr(Op)".

For example, the indexical used in the n-queens problem is defined as:

```
X in -dom(Y) /\ -(dom(Y)+N) /\ -(dom(Y)-N)
```

which compiles to the instructions

```
 val_1       % N
 dom_0       % dom(Y)
  setsub     % dom(Y)-N
   compl     % -(dom(Y)-N)
 val_1       % N
 dom_0       % dom(Y)
  setadd     % dom(Y)+N
   compl     % -(dom(Y)+N)
 dom_0       % dom(Y)
  compl      % -dom(Y)
   inter     % -dom(Y) /\ -(dom(Y)+N)
    inter    % -dom(Y) /\ -(dom(Y)+N) /\ -(dom(Y)-N)
halt
```

3.2.2 Variables A finite domain variable is an instance of the constraint variable of AGENTS, thus represented as:

```
struct finDom {
  struct varMethod *method;
  id               *env;
  suspension       *asusp;
  suspension       *msusp;
  fd_suspension    *isusp;
  struct finDom    *next;
  id               *trailed;
  unsigned long    info;
  unsigned long    min;
  unsigned long    max;
  bitmask          *d;
}
```

The `msusp` field contains a list of member constraints (see below) that are suspended on the variable.

The `isusp` field contains a list of suspension records { `int prop; indexical *f;`}, where `prop` encodes what must be pruned of the variable to reexecute the indexical `f`. That is, `prop` can either be `DOM`, meaning that any pruning of the variable will wake `f`, `MIN` (`MAX`), meaning that pruning the minimum (maximum) value of the variable will wake `f`, or `MINMAX`, meaning that any pruning which does involve *either* the minimum or the maximum value will wake `f`.

Furthermore, `next` is used for queueing the variable into the propagation queue when pruned, hence propagation is done in constant space, and `trailed` is the identity of the store where the variable was last trailed (initially set to NULL). `info` contains information about whether the variable is currently an interval or a set, what the last pruning consisted of, i.e. whether the maximum value, the minimum value, both, or neither was pruned, when the last pruning occurred, and how large the representation of the domain is. `min` (`max`) contains the current minimum (maximum) of the variable, and `d` contains an array of bitvectors representing the domain (set to NULL if the variable is an interval).

3.2.3 Constraints To interface FD with AGENTS we also need to define the constraints necessary to maintain the hierarchy of stores. There are two such constraints defined: the indexical constraint and the member constraint.

The *indexical constraint* is defined as:

```
struct fd_constraint {
  constraintMethod  *method;
  struct constraint *next;
  fd_indexical      *f;
}
```

simply encapsulating the indexical with the necessary interface functions. *Installing* an indexical constraint is done by applying the solver to the referred indexical in the store being installed. *Deinstalling* an indexical constraint is a void operation, and *promoting* an indexical constraint moves the constraint to the store being promoted into.

The *member constraint* is used for declaring a local pruning of some variable, and is defined as:

```
struct member_constraint {
  constraintMethod  *method;
  struct constraint *next;
  Term              X;
```

```
  unsigned long     info;
  unsigned long     min;
  unsigned long     max;
  id                *trailed;
  bitmask           *d;
}
```

where `X` is the variable being pruned, and the other fields derive their meaning from the finite domain structure. When *installed* the member constraint contains an image of the external content of `X` and `X` is marked as trailed in the local store, and when *deinstalled* the constraint contains an image of the local content of `X`. When *promoting* a member constraint, suspensions on the variable contained by the constraint are checked if they are affected by the promotion. For example, a sibling store becomes a child store after the promotion, and hence it may need to be installed.

4 The FD solver

Basically, the FD solver is an efficient implementation of the decision table Table 1. Whenever it cannot be decided whether an indexical x **in** r is entailed or inconsistent, the indexical acts like a reactive agent, suspending until more information is added to the store, thereby reexecuting. If the indexical is monotone, x is forced to be a subset of r, and hence domain prunings are propagated.

4.1 Solver optimizations

The solver exploits important optimizations, some of which are used in clp(FD) as well [DC93]. We now outline the ones specific to AKL(FD).

4.1.1 Equivalence marking Indexicals, known to be logically equivalent, are connected by references to a common flag (Section 3.2.1). Whenever one of the indexicals is decided entailed, the flag is set. Hence, before any indexical is executed the associated entailment flag is checked and if set the indexical is ignored.

4.1.2 Locality testing Before executing an indexical its locality is computed. Only if the indexical belongs to the environment or to the local store it should be executed. In other cases the indexical may belong to a child store, hence that store must be installed before executing the indexical, the indexical may belong to a sibling store, and hence it should be ignored, or the indexical belongs to an inconsistent store elsewhere in the hierarchy, and hence it should be dismissed.

By marking the path from the root of the hierarchy to the current store, checking whether a given store is an *ancestor*, which is a common case, of the current local store can be done efficiently.

4.1.3 Entailment checking The entailment of a finite domain constraint is best checked using logical conditions. It has been proven that antimonotone indexicals can be used for expressing such logical conditions [CCD94].

Hence, we use antimonotone indexicals to efficiently check entailment of finite domain constraints. This means that certain indexicals are marked as entailment checking, and treated correspondingly by the solver (see below). The constraint compiler of AKL(FD) thus compiles a constraint used for entailment checking into an antimonotone indexical which, when entailed, implies that the constraint is entailed as well (see Section 5).

4.2 Outline of the solver

The solver is based on an arc-consistency algorithm for indexicals. We now present the basic solver. For a full description see elsewhere [Car95].

Let Q be a finite queue/set of variables, and let σ be a constraint store.

Algorithm 4.2:

```
aklfd_check(Q, σ)
{
  while Q not empty do {
    set Q = Q \ {y};
    let F be the set of indexicals suspended on y;
    for each f ≡ x in r in F do {
      if f marked entailed then dismiss f and continue;
      if f marked dead then dismiss f and continue;
      let l be the locality of f;
      case l of {
      DEAD:
        mark f as dead and continue;
      CHILD:
        install l;
      SIBLING:
        continue;
      LOCAL:
      ANCESTOR:
        if f not affected by the pruning of y then continue;
        let I = x_σ ∩ r_σ;
        case I of {
        ∅:
          if r monotone in σ then fail else continue;
        x_σ:
          if r antimonotone in σ then mark f as entailed and dismiss it;
          continue;
        otherwise:
```

```
            if r not monotone in σ then continue;
            set σ = σ ∪ {x ∈ I};
            if x queued then update type of pruning of x;
            set Q = Q ∪ {x};
            wake all member suspensions to stores below l;
            if x determined then
            wake all agent suspensions in stores below or equal to l;
            if r constant in σ then mark f as entailed and dismiss it;
          }
        }
      }
    }
    return σ;
}
```

When σ is updated with $x \in I$ either a member constraint is added to σ (Section 3.2.3), or the update to x is trailed (Section 4.4). This choice is made by the labeling procedure.

The complexity of the algorithm depends on the number n of variables, the maximum domain size m of any variable, the maximum number e of suspended indexicals of any variable, the cost c of evaluating a range, and the number s of stores in the hierarchy.

Complexity 4.1 *The algorithm is $O(mnecs)$ in time [Car95].*

Note that in the AGENTS-implementation, installing l is postponed until the propagation in σ has terminated successfully. However, it is important to note that propagation between different levels of stores is performed. That is; local prunings of external variables are hidden by member constraints, and external prunings of locally suspended indexicals install the local store.

4.3 The emulator

The FD emulator, used for evaluating indexical ranges, is a stack machine with instructions corresponding to each FD term (Section 3.2.1). The emulator is run on a code sequence and an environment, and produces a set of natural numbers (represented by a bitvector or an interval).

Each FD operator, such as $..$, $\wedge$, $\vee$, $+$, $-$, etc, has a corresponding emulator instruction, which evaluates its arguments into a bitvector or an interval. For example the instruction for $\wedge$ replaces the two topmost sets on the stack with their intersection, either computed bitwise or by interval reasoning. The code for $\mathbf{dom}(X)$ pushes a set with a reference to the representation of X, where X is accessed as an offset on the emulator stack.

The control flow is sequential except for $r \Rightarrow r'$ which is evaluated as: first the code for r is evaluated, followed by an instruction `check` which

tests whether the top of the stack contains the empty set. If the test is true, a jump is made, passing the code for r'. If the test is false, the instructions for r' are evaluated.

4.4 Trailing of finite domain information

We have tried a scheme of trailing changes to the state instead of copying parts of the computation state. However, information concerning guards is currently not trailed, but only destructive changes to finite domains and indexicals are.

The fields `next`, `trailed`, `info`, `min`, `max`, `d` of a finite domain variable are trailed when the variable is pruned. The fields `ent` and `moninfo` of an indexical are trailed when the indexical becomes entailed or when its monotonicity information is updated.

The enumeration procedure for trailing is written in C, and uses two stacks; one choice-point stack and one trail stack. Each choice-point frame contains the variable currently selected for assignment, the next assignment value, the list of the rest of the variables to be enumerated, and a pointer to the top of the trail as it was before the variable was assigned its current value.

Guard suspensions from the variables to be enumerated are collected before the enumeration is initiated. Consequently, a solution is computed and a copy is made of the local computation state containing the solution. The solution is then promoted, and the collected guard suspensions are waked. When selecting the noncopied part of the computation state, the topmost choice-point frame is used for resetting the part of the trail corresponding the latest choice, and enumeration is reexecuted. The AKL part of the enumeration is as follows, where "?" is the guard operator which nondeterministically selects a clause:

```
labeling(L) :-
   collect_suspensions(L, S),
   init_labeling(L, C),
   c_labeling(C),
   labeling(C, S).

labeling(_, S) :-
    ? wake_suspensions(S).
labeling(C, S) :-
    ? c_labeling(C),
      labeling(C, S).
```

The parameter `C` denotes the bottom of the choice-point stack, such that several enumerations can be active simultaneously.

5 FD compiler

Indexicals are compiled with the rest of the AKL code. For each indexical `X in` r in the code, `akl_in(X,L,F,env(...),prop(...),mon(...),amon(...))` is called, where `L` is the index to the byte code for r, `F` is a variable used as entailment flag (see Section 3.2.1), `env(...)` contains the calling environment of r, i.e. the variables that occur in r, `prop(...)` contains the propagation dependencies between the environment and the indexical, `mon(...)` contains the variables that must be determined to make `X in` r monotone, and similarly for `amon(...)` and the case of the indexical being antimonotone.

The variables in `env(...)` are referred to through offsets, and we use special versions of the range functions for accessing the first three arguments, e.g. `dom_0` which computes the domain of the first argument.

The FD compiler compiles arithmetic finite domain constraints depending on whether the constraints are used for consistency or entailment checking. The compiler translates a constraint used for consistency checking either to a set of monotone indexicals, or to a conjunction of calls to builtin library constraints defined in terms of indexicals.

For example, the constraint $5x + y = 4z$ is either translated into

$$\texttt{'ax+y=t'}(5, x, y, t), \texttt{'ax=t'}(4, z, t),$$

where `'ax+y=t'`/4 and `'ax=t'`/3 are defined by a composition of indexicals, or to the indexicals

$$\begin{aligned} x \textbf{ in } & (4 * \mathbf{min}(z) - \mathbf{max}(y))/5..(4 * \mathbf{max}(z) - \mathbf{min}(y)/5), \\ y \textbf{ in } & 4 * \mathbf{min}(z) - 5 * \mathbf{max}(x)..4 * \mathbf{max}(z) - 5 * \mathbf{min}(x), \\ z \textbf{ in } & (5 * \mathbf{min}(x) + \mathbf{min}(y))/4..(5 * \mathbf{max}(x) + \mathbf{max}(y))/4. \end{aligned}$$

The choice is made depending on the arity of the constraint compiled.

If a constraint is used for entailment checking the compiler generates one antimonotone indexical which, when entailed as by Table 1, implies that the constraint is entailed.

Consider the example

$$c \equiv 2x + 3y \geq 5.$$

A suitable condition which implies the truth of c is derived by replacing $2x + 3y$ with its lower bound [CCD94]. Thus, we obtain the antimonotone indexical

$$5 \textbf{ in } 0..(2 * \mathbf{min}(x) + 3 * \mathbf{min}(y)).$$

The general translation for entailment is found in [Car95].

Also, two schemes are being developed which compile arbitrary propositional combinations of finite domain constraints into sets of indexicals and into sets of conditional clauses respectively. The compilation schemes for disjunction are described in detail elsewhere [CC95].

6 Conclusion

We describe the implementation of AKL(FD). The generic constraint interface, supporting hierarchical constraint stores, of AGENTS is explained, as well as its instantiation with FD. The FD solver, which exploits important optimizations, and supports both solving and checking constraints, is outlined. Finite domain constraints are compiled in AKL(FD), and fundamental techniques used by the compiler are examplified.

References

[Car95] Björn Carlson. *Compiling and Executing Finite Domain Constraints.* Uppsala Theses in Computing Science 21, Uppsala University, May 1995. (Also available as SICS Dissertation Series 18.).

[CC95] B. Carlson and M. Carlsson. Compiling and Executing Disjunctions of Finite Domain Constraints. In *Proceedings of the Twelfth International Conference on Logic Programming.* MIT Press, 1995.

[CCD94] B. Carlson, M. Carlsson, and D. Diaz. Entailment of finite domain constraints. In *Proceedings of the Eleventh International Conference on Logic Programming.* MIT Press, 1994.

[CJH94] B. Carlson, S. Janson, and S. Haridi. AKL(FD): a concurrent language for finite domain programming. In *Logic Programming: Proceedings of the 1994 International Symposium.* MIT Press, 1994.

[DC93] D. Diaz and P. Codognet. A Minimal Extension of the WAM for clp(FD). In *Proceedings of the International Conference on Logic Programming*, 1993.

[Jan94] Sverker Janson. *AKL—A Multiparadigm Programming Language.* Uppsala Theses in Computing Science 19, Uppsala University, June 1994. (Also available as SICS Dissertation Series 14.).

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*, San Diego, California, October 1991. MIT Press.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Symposium on the Principles of Programming Languages.* ACM/SIGPPLAN, 1991.

[VHSD91] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Computer Science Department, Brown University, 1991.

[VHSD92] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint Logic Programming over Finite Domains: the Design, Implementation, and Applications of cc(FD). Technical report, Computer Science Department, Brown University, 1992.